

SYSCALL WRAPPERS

CS3500 Project Report - Group 3

File I/O wrappers

Contributors - Yashvardhan Toshniwal, Kritang Kothari, Daksh Sehra

Introduction

File I/O wrappers are custom implementations designed to enhance the functionality, security, and performance of standard file input/output operations in the Linux kernel. By wrapping around system calls such as open, read, write, and close, these wrappers provide additional features like logging, access control, error handling, and performance optimization.

Implemented Wrappers

1. Logger

- **Implementation:**
 - Functions: open_logger, read_logger, write_logger, close_logger.
 - Logs messages to a specified log file for each file operation.
 - Uses dlsym to call the original system calls and logs the results.
- **Advantages:**
 - Helps in debugging.
 - Monitors application behavior.

2. Safe Read

- **Implementation:**
 - Function: safe_read.
 - Checks file statistics and available bytes before performing the read operation.
 - Uses fstat and ioctl to gather file information and ensure safe reading.
- **Advantages:**
 - Gives warning when reading from empty files.
 - Checks if the requested byte count is greater than the available bytes. If it is, it asks the user whether to read all the remaining bytes
 - Ensures robust error handling for read operations.

3. Safe Open

- **Implementation:**
 - Function: safe_open.
 - Validates flags and checks file existence before opening.
 - Uses custom validation functions and prompts the user for actions if necessary.
- **Advantages:**
 - Prevents common file handling errors.
 - Ensures robust error handling.

4. Access Control and Permissions Enforcement Wrapper

- **Implementation:**
 - Function: check_permission.
 - Verifies user permissions before allowing file access.
 - Uses getuid and getpwuid to check the current user's identity against allowed users.
- **Advantages:**
 - Enhances security by preventing unauthorized access.

5. Rate-Limiting

- **Implementation:**
 - Functions: rate_limited_open, rate_limited_read.
 - Uses a RateLimiter structure to track and limit operations within a specified time window.
 - Implements rate_limiter_init and rate_limiter_check to manage rate limiting.
- **Advantages:**
 - Protects against abuse.
 - Ensures fair resource usage.
 - Prevents denial-of-service attacks by throttling excessive access.

6. Buffer

- **Implementation:**
 - Functions: buffer_init, buffer_add, buffer_flush.
 - Manages a buffer structure to accumulate data before writing to the file.
 - Uses memcpy to add data to the buffer and flushes it when necessary.

- **Advantages:**
 - Improves performance by reducing the number of write operations.

7. **Master Wrapper**

- **Implementation:**
 - Functions: open, read, write, close.
 - Intercepts system calls and applies the appropriate custom wrappers.
 - Uses LD_PRELOAD to override standard library functions with custom implementations.
- **Advantages:**
 - Simplifies the management of individual wrappers.
 - Provides a single point of control for various file I/O enhancements.

Possible Customizations:

1. **Rate Limiting Customizations**
 - a. **Adjust Rate Limits:** Modify the MAX_OPERATIONS and TIME_WINDOW constants in rate_limiting.h to change the rate limit parameters.
 - b. **Different Rate Limiters for Different Operations:** Implement separate rate limiters for different file operations or files.
2. **Permission Control Customizations**
 - a. **Custom Access Control Policy:** Change the logic in check_permission function in control_permission.c to enforce custom access control policies.
 - b. **Check for Specific Groups:** Extend the check_permission function to check for specific user groups.
3. **Buffer Customizations**
 - a. **Adjust Buffer Size:** Change the BUFFER_SIZE constant in buffer.h to modify the size of the buffer.
 - b. **Custom Buffer Flushing Logic:** Modify the buffer_flush function in buffer.c to implement custom logic for flushing the buffer.
4. **General Customizations**
 - a. **Change Log File Location:** Modify the LOG_FILE constant in logger.h to change the location of the log file.
 - b. **Customize Log Messages:** Update the log messages in logger.c to include additional information or change the format.
 - c. **Customized Error Handling:** Can modify code behaviour to alert or handle specific error messages.

- The aforementioned wrappers have been implemented as a user space library. To make use of these wrappers use the Makefile to compile them into a shared object file, and run your executable with the command LD_PRELOAD=./filewrp.so ./<exec_name>

- We also tried implementing these in kernel space as well. Files can be found in fileio_kernel folder on drive.

- The wrappers already made were rewritten using Kernel APIs for the syscalls we were using previously, tried compiling it with the kernel but there were some compilation issues.

Memory Management Wrappers

Contributors - Raadhes Chandaluru, Dev Mehta, Mith Jain, Rohan Bagati

Allocation tracking in mmap

- **Implementation:**
 - A custom struct to track allocations for all the mmap calls, storing all the memory instances in a linked list, helpful for logging.
 - Detects various errors like EINVAL, EBADF, ENOMEN, etc
- **Advantages:**
 - Enhanced error handling: If mmap fails, it reports errors and terminates safely.
 - Improves debugging and performance analysis through detailed logging of memory usage.

Safe Memory Unmapping

- **Implementation:**
 - Structure: Using the data structures from safe_mmap, we traverse the alloc_list and unmap chunk by chunk while checking for errors for precise reporting.
 - Detection: Detects errors and reports chunks in which error is generated.
- **Advantages:**

- Prevents undefined behavior by ensuring `munmap` is only called on valid memory regions.
- Makes memory management more robust by integrating error handling and tracking updates.

Debugging Dynamic Memory Methods

- **Implementation:**
 - Structure: a linked list to track each allocation, storing the pointer, size, and a reference to the next allocation record.
 - Auxiliary Functions: `track_allocation` function adds a new `AllocRecord` added to the front and `untrack_allocation` function removes a specific record when memory is freed.
 - Detection: `detect_memory_leaks` iterates through the linked list and prints any allocations that were not freed. If no leaks are detected, it reports that no leaks were found.
- **Advantages:**
 - Simplifies memory debugging by automatically logging every memory operation.
 - Detects and pinpoints memory leaks, reducing the risk of out-of-memory errors in long-running programs.
 - Provides clear visibility into memory usage patterns, aiding performance tuning.

Debugging Dynamic Memory Syscalls

- **Implementation:**
 - Function `brk`
 - Uses a `brk` mutex for ensuring multiple threads do not access the heap memory at once leading to inconsistencies.
 - Upon failure of `brk`, it checks if `mmap` call returns an error, if not it allots the memory and returns 0, wrapping around the `brk` failure.
- **Advantages:**
 - Thread safety, so multi-threaded heap allocations do not lead to inconsistency, in case `ptmalloc2` is not used.
 - Internally calls the `mmap` call also upon failure of the `brk` system call, for larger memory requests.

File Descriptor Management

- **Implementation:**
 - This code defines a file management system that tracks open file descriptors in a global structure using a custom structure **OpenFileManager** to track open file descriptors.
 - The **open_and_track** function dynamically allocates or resizes the descriptor array, opens the file, and stores the descriptor, while **close_and_untrack** searches for and removes a descriptor from the array before closing the file.
 - The **close_all_files** function iterates over the array to close all open files and resets the count, ensuring proper resource management and avoiding file descriptor leaks.
- **Advantages:**
 - Simplifies resource management, particularly in programs with many open files.
 - Prevents resource leaks by enforcing proper closure of all file descriptors at program exit or error recovery.
 - Improves program stability by preventing exhaustion of file descriptor limits.

Memory Pool Call

- **Implementation:**
 - The `init` function instantiates a pool of memory using `malloc` at once, with size and number of blocks in the pool specified.
 - A struct called `PoolInstance` stores the starting source location of the pool, the list of free locations and helps counting variables.
 - As and when memory is requested, the pool is checked for free location and allotment is done.
- **Advantages:**
 - Very helpful for programs that frequently access small chunks of memory one by one, saves the loaf of repeated system calls.
 - Helps to improve reference of locality as memory is allocated from a contiguous pre-allocated pool, reducing cache misses.

Aligned Memory Allocation and Deallocation

- **Implementation:**
 - **Alignment Handling:** Ensure the alignment is a power of 2 and at least the size of a pointer, then calculate an aligned memory address by rounding up the base address using bitwise operations.
 - **Memory Reservation:** Use to allocate extra memory to accommodate alignment adjustments, and store the original base address just before the aligned memory for correct deallocation.

- **Deallocation Safety:** Retrieve the original base address during and calculate the total allocated size to safely and correctly release the memory using.
- **Advantages:**
 - Ensures compatibility with systems or libraries requiring aligned memory (e.g., for SIMD instructions or hardware page alignment).
 - Improves performance by reducing the overhead of misaligned memory accesses.
 - Simplifies deallocation by correctly identifying the base address of the allocated region, preventing potential errors.

Process Wrappers

Contributors - Aditya Srivastava, Aditya Jain, Harsh Vardhan Daga, Shreyanshu Gurjar

Process Pool Manager

This Process Pool Manager is a wrapper that manages a pool of worker processes. It allows tasks to be distributed efficiently across multiple processes, much like a thread pool but using processes for better isolation and utilization of multi-core systems.

- **Implementation**
 - **Dynamic Task Queue:** Accepts tasks dynamically and distributes them to workers (different child processes) created using `create_worker(index)` function that further uses `fork()` system call inside it.
 - **Inter-Process Communication (IPC):** Each worker (child process) shares a pipe with the parent process, with the help of which, the parent process allots tasks to each child process to complete. The parent process adds tasks to the write end of the pipe and the workers read from the read end of their corresponding pipe.
 - **Task Scheduling:** Currently, the tasks are assigned to workers in a cyclic fashion, one after the other, but it can further be customized as per the priorities of different tasks.
 - **Graceful Shutdown:** Ensures all tasks are completed before shutting down.
- **Advantages**
 - Reduces overhead by reusing existing worker processes, avoiding frequent process creation and destruction.
 - Maximizes CPU core usage by running processes in parallel.

Process Graph Deadlock Detection

To handle potential deadlocks, we create wrappers for system calls related to resource allocation(mutex and semaphore). These wrappers dynamically build a process-resource dependency graph, detecting circular waits and, thus, potential deadlocks.

- **Implementation Details**
 - **Resource Allocation Wrappers:** System calls like `sem_wait()` and `pthread_mutex_lock()` are wrapped to log resource requests and releases into a dependency graph.
 - **Graph Construction:** The process-resource dependency graph is dynamically updated whenever a process/thread interacts with resources.
 - **Cycle Detection:** A Depth-First Search (DFS) algorithm runs periodically at every allocation on the graph to detect circular dependencies indicating potential deadlocks.
 - **Deadlock Resolution:** Upon detecting a cycle, the wrapper automatically kills the most recent process involved in the deadlock to resolve it.
- **Advantages**
 - **Real-Time Detection:** Enables continuous monitoring of process-resource dependencies, allowing immediate detection of potential deadlocks.
 - **Minimal Overhead:** Wrappers efficiently integrate with existing system calls, introducing minimal performance impact during process execution.

Process Cloaking Wrapper

To create child processes that are hidden from common process monitoring tools like `ps` and `top` by filtering them out during the `read()` syscalls made by `ps` and `top`. Thus effectively “cloaking” them.

- **Implementation Details**
 - **Fork Wrapping:** The `fork()` system call is wrapped to add new child processes to a hidden process list stored in a file.

- **Intercepting read():** The `read()` system call is hooked to filter out the hidden processes when commands like `ps` or `top` attempt to read from `/proc`.
 - **File-Based Cloaking:** A list of hidden processes is maintained in a file, checked during each `read()` to remove entries corresponding to cloaked processes.
- **Advantages**
 - **Enhanced Process Privacy:** Prevents specific processes from being monitored through common tools like `ps` or `top`, improving confidentiality.
 - **Increased Security:** Reduces the risk of detection by attackers or unauthorized users, useful for sensitive tasks or security-focused applications.

Wait for All Children wrapper

The wrapper `wait_for_all_children` when called with a parent `pid`, it blocks the parent process until all the child processes exit.

- **Implementation**
 - The file `/proc/<PID>/task/<TID>/children` contains the child PIDs of a given thread or process.
 - Iterate over child processes and call `waitpid()` on all of them.
 - Hooked on `waitpid`, to be called as `waitpid(parent_pid, status, options)`
- **Advantages**
 - **Prevent Zombie Processes:** Ensures no child process is left in a zombie state. A process doesn't need to call `waitpid()` for every child process it creates as this wrapper automates calling `waitpid()` on all child processes.

Comprehensive IPC Logging Wrappers

A collection of logging wrappers for various IPC mechanisms such as pipes, message queues, shared memory, and semaphores. These wrappers log key information about the system calls without altering the underlying application logic. The logs provide useful insights into the operations, parameters, and results of the IPC calls, improving debugging and monitoring.

- **Implementation**
 - Hooks on IPC-related system calls like `pipe()`, `msgsnd()`, `msgrcv()`, `shmget()`, and `semop()` using function interception (e.g., `dlsym` with `LD_PRELOAD`).
 - Logs are generated by wrapping each system call to record input parameters, timestamps, and results.
 - Data is stored in log files with clear identifiers for each process and IPC mechanism to track interactions across the system.
- **Advantages**
 - **Comprehensive Auditing:** Provides a complete log of all Inter-Process Communication (IPC) events, enhancing traceability and accountability in the system.
 - **Security Monitoring:** Helps identify unauthorized or suspicious IPC activity, improving overall system security.

Priority Enforcement wrapper

The Priority Enforcement Wrapper extends the `fork()` system call to set CPU and I/O priorities for child processes. This helps ensure that high-priority tasks receive sufficient resources while low-priority tasks don't hinder overall system performance (Up for discussion with TA).

- **Implementation**
 - After calling `fork()`, the wrapper sets the CPU and I/O priorities for the child process.
 - CPU priority is set using `setpriority(PRIO_PROCESS, 0, nice_value)`.
 - I/O priority is set using a system command: `system("ionice -c <io_class> -n <priority> -p <pid>")`.
- **Advantages**
 - **Controlled Resource Usage:** Lower-priority tasks run without impacting high-priority applications.
 - **Dynamic Resource Allocation:** Easily adjust priorities at runtime to adapt to system load.

Process Monitoring and Logging Wrapper

This wrapper keeps track of details such as start time, end time of the process. It also keeps a log of resource usage statistics such as CPU time, memory usage, context switches and the exit status of the process.

- **Implementation**
 - Hooked on `execvp()`, to be called as `execvp(const char *program, char *const argv[])`.
 - Uses `fork()` to create a child process.
 - The child process runs the given program, while the parent process monitors resource usage and other statistics using `getrusage`.
- **Advantages :**
 - Helps detect abnormal process behaviour, such as crashes or long execution times.

Custom Waitpid Wrapper

The **Custom Waitpid Wrapper** extends the `waitpid()` system call to enhance the monitoring and handling of child processes. This

wrapper allows the parent process to track the status of long-running child processes without blocking, implement timeouts to prevent indefinite blocking, and provides detailed information on the exit status of the child process.

- **Implementation**
 - After invoking `waitpid()`, the custom `waitpid` dynamically loads the original `waitpid` function using **`dlsym`**.
 - **Non-blocking Wait:** The parent process enters an infinite loop, repeatedly checking the child's status using the original `waitpid` with **`WNOHANG`** option.
 - **Timeout:** If the child doesn't exit within the specified timeout time, then the parent kills the child process.
 - Sleep is also added in each iteration so that the parent does not check the status continuously in every iteration, helping in reducing the CPU usage.
- **Advantages**
 - **Improved Monitoring:** Tracks and provides detailed insights into child process statuses.
 - **Timeout Handling:** Implements timeouts to avoid indefinite blocking of long-running processes.

Zombie Process Logger

This wrapper extends the `kill()` and `waitpid()` system calls to provide enhanced process management and logging. This wrapper is designed to log any zombie processes when these syscalls are invoked. It tracks the lifecycle of child processes, ensuring that no zombies remain after termination, and logs relevant information for debugging and process management.

- **Implementation**
 - The wrapper intercepts calls to `kill()` and `waitpid()` to manage the termination of processes.
 - This wrapper is based on the idea that zombie processes might be the siblings of the processes being called to end using `waitpid` or `kill` syscalls.
 - In case of `waitpid()`, the wrapper logs the process termination status(normal, signal or stopped).
 - In case of `kill()`, the wrapper logs whether the signal was sent successfully to the process or if it failed. In case of error, a specific error message is logged as well.
 - Finally, an infinite loop is used which checks for and cleans up any zombie process by calling the `waitpid(-1, &status, WNOHANG)`, logging the status of each terminated process.
- **Advantages**
 - **Zombie Process Logging:** Logs details about any zombie processes, making it easier to track and debug process termination issues.
 - **Process Cleanup:** Ensures that processes are properly terminated, avoiding the creation of zombie processes.

Custom system calls and wrappers at kernel level

Working on ubuntu 22.04.5 virtual machine

Downloaded kernel from

<https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.0.7.tar.xz>

(Other versions with respective URL)

How to compile and install

1. Set the configurations using `make menuconfig` or copy previous kernel configs to `.config` file. (`cp -v /boot/config-$(uname -r) .config`) in linux-version folder
2. Use `make` command to compile. Wait several hours. (Used `make -jn` where `n` is number of processors to do faster)
3. Use `make modules_install` and `make install` to add the kernel to grub.
4. Can boot with new kernel using grub. It will be visible in advanced options in grub.
Once booted check kernel version with `uname -r` command in terminal.

Notes :Compilation of 4.17.4 failed due to some error. Possibly due to reverse an incompatibility error in gcc.

Custom System call created as demonstration

- System call code:

```
root@ubunt1:/home/rad/linux-6.0.7/hello# cat hello.c
#include<linux/kernel.h>

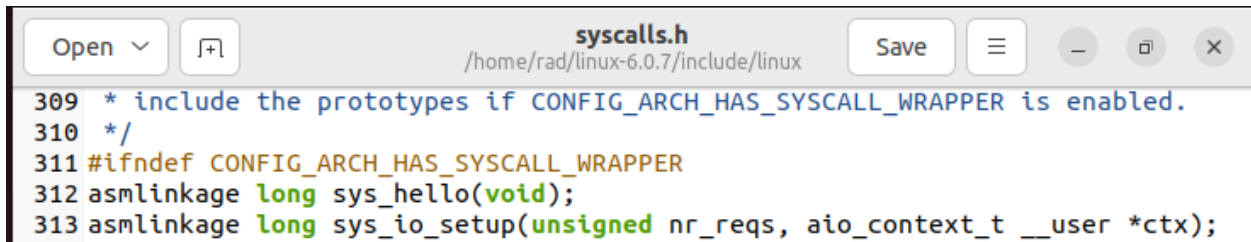
#include<linux/syscalls.h>

SYSCALL_DEFINE0(hello){
    pr_info("My system call is running\n");
    return 0;
}
```

- System call Makefile:

```
root@ubunt1:/home/rad/linux-6.0.7/hello# cat Makefile
obj-y := hello.o
```

- Updated syscalls.h with system call declaration:



```
syscalls.h
/home/rad/linux-6.0.7/include/linux
Save

309 * include the prototypes if CONFIG_ARCH_HAS_SYSCALL_WRAPPER is enabled.
310 */
311 #ifndef CONFIG_ARCH_HAS_SYSCALL_WRAPPER
312 asmlinkage long sys_hello(void);
313 asmlinkage long sys_io_setup(unsigned nr_reqs, aio_context_t __user *ctx);
```

- Added system call at end of system call table with new system call number

```
415 545      x32      execveat      compat_sys_execveat
416 546      x32      preadv2       compat_sys_preadv64v2
417 547      x32      pwritev2      compat_sys_pwritev64v2
418 548      common   hello         sys_hello
419 # This is the end of the legacy x32 range.  Numbers 548 and above are
420 # not special and are not to be used for x32-specific syscalls.
```

- Compile kernel as done previously and update grub. Reboot system and use new kernel. (Check advanced options) Use 6.0.7 which is the latest version.
- Do note that you must press shift before original kernel boots to get grub menu



```
GNU GRUB  version 2.06

*Ubuntu, with Linux 6.8.0-49-generic
Ubuntu, with Linux 6.8.0-49-generic (recovery mode)
Ubuntu, with Linux 6.8.0-40-generic
Ubuntu, with Linux 6.8.0-40-generic (recovery mode)
Ubuntu, with Linux 6.0.7
Ubuntu, with Linux 6.0.7 (recovery mode)
Ubuntu, with Linux 6.0.7.old
Ubuntu, with Linux 6.0.7.old (recovery mode)
```

- Test the system call by writing and compiling a C code to call the syscall by number.
- Custom System call running linux 6.0.7 kernel ubuntu distribution: (check system logs to see if it works). Also returns 0 which means the system call is running.

```
root@ubunt1:/home/rad/Documents# cat testsyscall.c
#include<stdio.h>
#include<sys/syscall.h>
#include<unistd.h>

int main(void){
    printf("%ld\n", syscall(548));
    return 0;
}
root@ubunt1:/home/rad/Documents# gcc testsyscall.c
root@ubunt1:/home/rad/Documents# ./a.out
0
root@ubunt1:/home/rad/Documents# dmesg | tail
[ 24.612279] rfkill: input handler enabled
[ 27.153658] ISO 9660 Extensions: Microsoft Joliet Level 3
[ 27.154494] ISO 9660 Extensions: RRIP_1991A
[ 27.350965] rfkill: input handler disabled
[ 28.279085] kauditd_printk_skb: 8 callbacks suppressed
[ 28.279089] audit: type=1400 audit(1732807452.416:81): apparmor="DENIED" operation="capable" profile="/usr/lib/snapd/snap-confine" pid=1795 comm="snap-confine" capability=12 capname="net_admin"
[ 28.279318] audit: type=1400 audit(1732807452.417:82): apparmor="DENIED" operation="capable" profile="/usr/lib/snapd/snap-confine" pid=1795 comm="snap-confine" capability=38 capname="perfmon"
[ 185.737567] My system call is running
[ 289.703725] My system call is running
[ 300.799802] My system call is running
root@ubunt1:/home/rad/Documents#
```


Hello syscall implemented in linux 5.19

```
rad@ubuntu1:~/Documents$ cat testsyscall.c
#include<unistd.h>
#include<sys/syscall.h>
#include<stdio.h>

int main(){
    int id = syscall(548);
    printf("%ld\n", id);
    return 0;
}
```

```
ration="capable" profile="/usr/lib/snapd/snap-confine"
ine" capability=38 capname="perfmon"
[ 40.551169] This is hello syscall in linux 5.19
root@ubuntu1:/home/rad/Documents# SSS
```

Disable custom system call in a particular process

- In this extension we look to disable the hello system call in the root process of a pid namespace. (i.e the process who pid is 1). The goal is that this should be done purely in the kernel context WITHOUT any userspace code.
- This involved finding the implementation of getpid to figure out the kernel function to call to get the process pid. It should be noted that system calls are not usually called directly in the kernel.

Modified system call code:

```
SYSCALL_DEFINE0(hello) {
    int retval = task_tgid_vnr(current);
    if ( retval != 1 ) {
        pr_info("This is hello syscall in linux 5.19 from non - root process.\n");
    }else {
        pr_info("Cannot run syscall as this is root process in its pid namespace.\n");
        return -1;
    }

    return 0;
}
```

Test program:

```
int main(){
    int x = unshare(CLONE_NEWPID);
    fork();
    int retsys = syscall(548);
    printf("Hello world, Im process %d. My custom syscall status: %d.\n",
getpid(), retsys);
    return 0;
}
```

Stdout:

```
root@ubuntu1:/home/rad/Documents# ./a.out
Hello world, Im process 2263. My custom syscall status: 0.
Hello world, Im process 1. My custom syscall status: -1.
root@ubuntu1:/home/rad/Documents# dmesg | tail
```

Kernel logs:

```
[ 59.200548] This is hello syscall in linux 5.19 from non - root process.
[ 59.200703] Cannot run syscall as this is root process in its pid namespace.
```

Resource Management (Disable fork in process and further children)

- Two system calls created to enable this feature
- Main points:
 - No user space data needs to be stored in any file, as all meta-data regarding disabling is stored in kernel data structures (Specifically the task_struct of the processes)
- syscall(549): fork_if_not_disable()
 - The fork_if_not_disable is a system call wrapper for the original fork implemented as a system call within the kernel.
- syscall(550): disable_fork()

- Updates kernel data structure for the process to disable further calls to `fork_if_not_disable()` in the same process or any of its descendants
- Future Work:
 - We can simply copy the system call code for `fork_if_not_disable` to the original `fork` so that the user experience does not change. Currently we have not implemented it as such as it is only a prototype.
- Use: This system call can be used to safely disable forks to prevent memory / resource overflow in use cases when users predicts fork to be called in incorrect places.

```
SYSCALL_DEFINE0(disable_fork) {
    current->can_fork = DO_NOT_FORK;
    return 0;
}

SYSCALL_DEFINE0(fork_if_not_disable) {
    if (current->can_fork == DO_NOT_FORK)
        return -1;
}

#ifdef CONFIG_MMU
struct kernel_clone_args args = {
    .exit_signal = SIGCHLD,
};

return kernel_clone(&args);
#else
/* can not support in nommu mode */
return -EINVAL;
#endif
return 0;
}
```

```
root@ubuntu1:/home/rad/Documents# ./a.out
First custom fork call: 3462
Custom fork call after disable: -1
First custom fork call: 0
Custom fork call after disable: -1
```

```
int disable_fork(){
    return syscall(550);
}

int fork_if_not_disable(){
    return syscall(549);
}

int main(){
    printf("First custom fork call: %d\n", fork_if_not_disable());
    disable_fork();
    printf("Custom fork call after disable: %d\n",
        fork_if_not_disable());
    return;
}
```

Close All Files System Call

- Created a close all files system call
- It should be noted that linux has a system call that closes files in a certain range
- This implementation was used as reference to create the close all files system call

Main Issues Encountered:

1. Compilation Issues which vary based on kernel version
2. Difference in system call implementation based on kernel version.
 - a. In higher versions `SYSCALL_DEFINE`(some fields) can be used, whereas in lower versions it is different.
 - b. Changes to Makefile differs based on kernel version. In higher versions may have to change Kbuild file whereas in other versions Makefile change is sufficient.
3. Compilation time is extremely long, generally exceeding 2 hours (Highly dependent on system specifications). Any small change in a file may lead to cascading dependencies that may cause the whole kernel to recompile.

References

- Common errors and required libraries specified in below references to compile smoothly
- 1. <https://phoenixnap.com/kb/build-linux-kernel> (Compiling kernel 6)
- 2. <https://medium.com/anubhav-shrimal/adding-a-hello-world-system-call-to-linux-kernel-dad32875872> (Kernel 4.17.4)
- 3. [https://medium.com/@aryan20/create-custom-system-call-on-linux-6-8-126edef6caaf#:~:text=Create%20the%20new%20syscall&text=%22SYSCALL_DEFINE\(\)%20macro%20rather%20than.for%20the%20parameters%20as%20arguments.%E2%80%9D&text=This%20is%20a%20prototype%20for%20the%20syscall%20function%20we%20created%20earlier.](https://medium.com/@aryan20/create-custom-system-call-on-linux-6-8-126edef6caaf#:~:text=Create%20the%20new%20syscall&text=%22SYSCALL_DEFINE()%20macro%20rather%20than.for%20the%20parameters%20as%20arguments.%E2%80%9D&text=This%20is%20a%20prototype%20for%20the%20syscall%20function%20we%20created%20earlier.) (kernel 6)

Attempt at implementing a wrapper library

- Tried to implement the previously made file i/o wrappers inside kernel code.
- Firstly we can't call some system calls directly as we had done in user space due to security reasons.
- So we used kernel APIs like `flip_open/flip_close`(for opening/closing a file), `vfs_read/write`, `set_fs`, `printk`, `kmalloc` etc. for the same functionality as the user space syscalls.
- Follow the same steps as above to create a new folder, put all the .c files here and add a Makefile like shown below.

```
open.c x Makefile x
1 obj-y := logger.o control_permission.o rate_limiting.o Safe_open.o
  Safe_read.o master_wrapper.o
2 |
```

- Add the syscalls into the syscall table and the syscalls.h file.
- Repeat all the steps and recompile the kernel.
- There were some issues with compilation due to some calls not being defined properly and unexpected behaviour.
- Some .c files did compile but I was not able to test it, as my kernel compiled but wasn't able to run OS onto it.
- Got the error: driver frame buffer was unable to register with bus_type coreboot ; bus was not initialized
- Searched google but wasn't able to fix the error.

System Call Hooks in the Project

As the name sounds, system call hooks intercept(hook) and modify system calls to extend or change their behavior. In this project, hooks are used to implement wrappers around critical system calls like `fork()`, `read()`, and `execvp()` to manage processes, track resources, and improve security.

Implementation

The project uses **user-space hooking** through the `LD_PRELOAD` mechanism:

1. **LD_PRELOAD** loads a custom shared library that overrides system calls.
2. The original system call is accessed using `dlsym()`, which dynamically resolves the address of the real syscall by returning the pointer to the next instance of the syscall in the table.
3. After getting the pointer to the syscall we can do some operation and then call the original syscall. Some more operations can follow this.

Considerations

- **Thread Safety:** When using hooks in a multi-threaded environment, care is taken to avoid race conditions, especially in functions like `waitpid()` and `fork()`, which may interact with multiple threads.
- **Security:** The hooking mechanism must be secure, ensuring that malicious behavior or unintended interference with system calls is avoided.
- **Performance:** Performance is carefully monitored since the hooks add extra function calls before and after the original system calls. Efficient use of `dlsym()` and other mechanisms ensures minimal overhead.

```
// Example Fork Hooking
pid_t fork(void) {
    static pid_t (*original_fork)(void) = NULL;
    if (original_fork == NULL) {
        original_fork = dlsym(RTLD_NEXT, "fork");
        if (!original_fork) {
            fprintf(stderr, "Error in dlsym: %s\n", dlerror());
            return -1;
        }
    }
    // Do whatever
    // Call the real fork
    pid_t pid = original_fork();
    // Do whatever
    return pid;
}
```

References

<https://keiran.scot/2022/10/10/Evasion-Techniques-Hiding-your-process-from-ps/> : used this blog to learn about hooks and implement a syscall wrapper using it,